Challenges of deploying eBPF-based tracing in embedded systems, and alternatives libtracefs & libtraceevent

Bean Huo (beanhuo@micron.com)

Goals of this talk

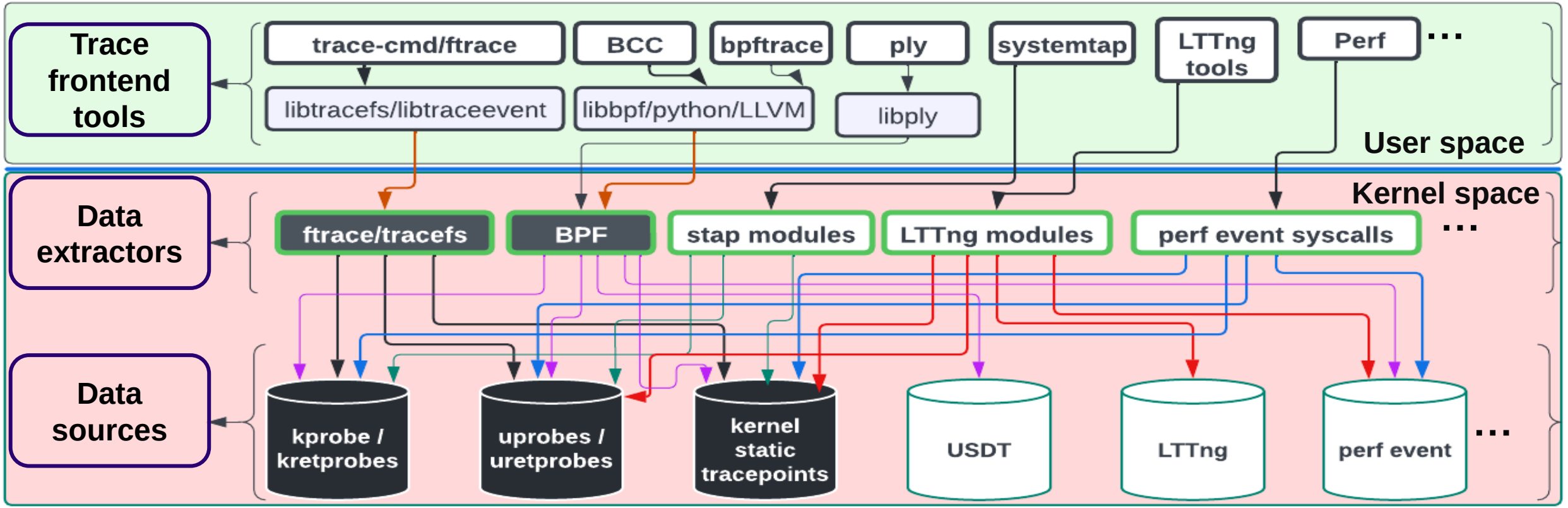
Share what we learned and the challenges/problems we encountered while using eBPF-based tracing tools in embedded systems.

Encourage discussion and participation in the Linux ecosystem community on eBPF portability further improvements, or finding alternatives for embedded systems, such as libtracefs / libtraceevent.

Agenda

- A brief introduction to Linux trace systems
- eBPF and Challenges of deploying it in embedded systems
- Libtracefs / libtraceevent
  - Libtracefs / libtraceevent based (ftrace/tracefs) and eBPF/libbpf based trace systems data flow
  - Example based on libtracefs/libtracevent
- Comparison between tools based on two different tracing infrastructures
  - system overhead, dependency, compilation
- Conclusion

Linux trace system



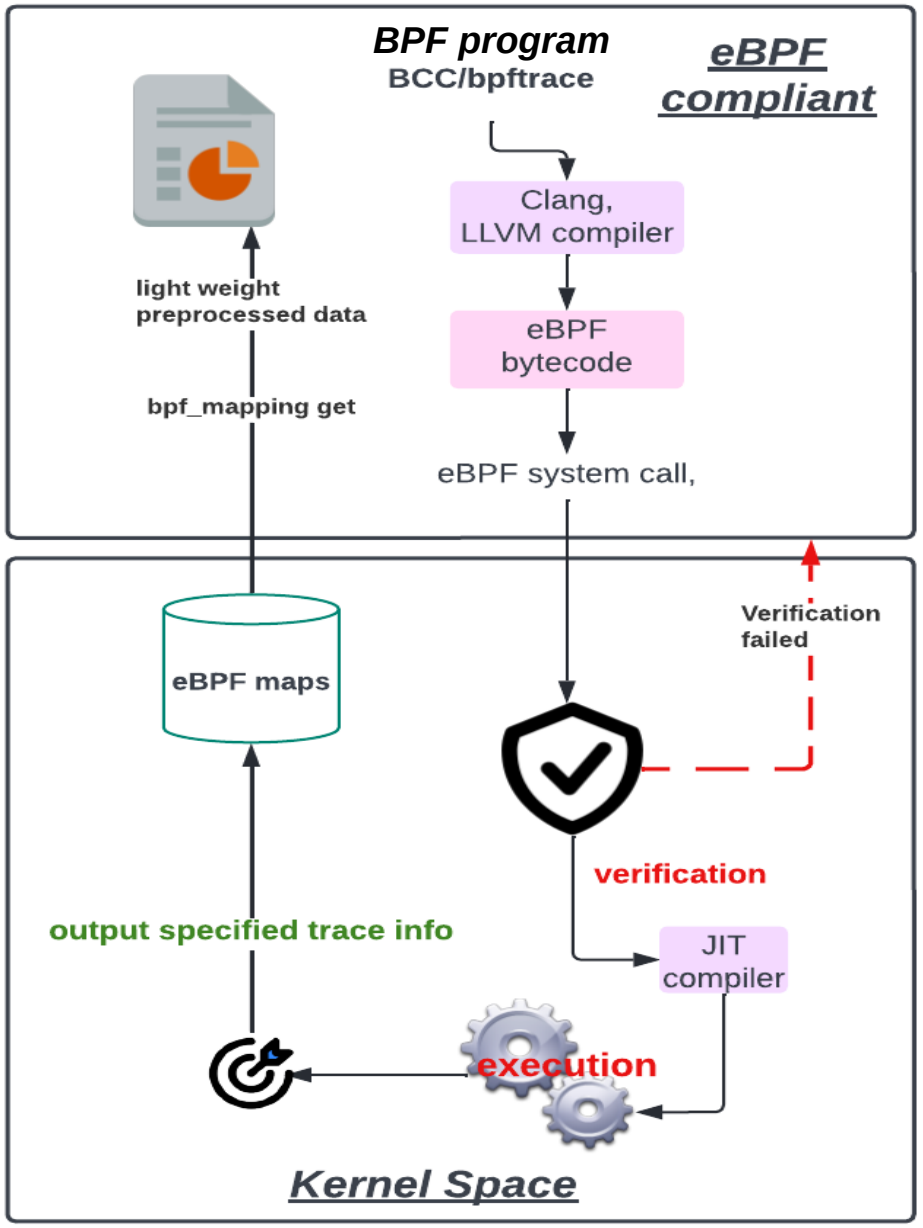
- Data sources (probes)
  - The places in the kernel that trace data is produced, and trace data extractors can obtain data from it
- Trace data extractors
  - Methods to get trace data and return the data to user space
- Trace frontend tools
  - Configure traces, attach Tracepoints/probes, organize trace data, and display it in a friendly way

BPF/eBPF

- Derived from BPF (Berkeley Packet Filter, used for network message filtering):
  - BPF has been redesigned to enable more options, data structures, which make BPF go beyond network message filters and become a Linux tracing infrastructure in the kernel.
  - Allows user to pass user space parameter to kernel and retrieve trace data from kernel space. That is eBPF (extended BPF).

- eBPF uses virtual machine (execution engine) in the kernel to execute bytecode (passed from user space) in a safe manner and efficient way.

- eBPF based tool developers can specify their own trace data and even design its own trace data print format before passing data to user space, getting the trace data in a lighter way.



Setup eBPF environment in Linux kernel

- *CONFIG_BPF=y*
- *CONFIG_BPF_SYSCALL=y*
- *CONFIG_BPF_JIT=y*
- *CONFIG_BPF_EVENTS=y* **(Kernel 4.7 and later)**
  - *CONFIG_KPROBE_EVENTS=y or CONFIG_UPROBE_EVENTS=y*
  - *CONFIG_FTRACE=Y*
  - *CONFIG_PERF_EVENTS=Y*
- *CONFIG_HAVE_BPF_JIT=y* ***(kernel 4.1 through 4.6)***
- *CONFIG_HAVE_EBPF_JIT=y* **(Kernel 4.7 and later)**
- CONFIG_DEBUG_INFO_BTF=y (**kernel 5.2 and later)**

  These new BPF binaries are only possible if this kernel config option is set. It adds about 1.5 Mbytes to the kernel image (this is tiny in comparison to DWARF debuginfo, which can be hundreds of Mbytes). Ubuntu 20.10 has already made this config option the default, and all other distros should follow. Note to distro maintainers: it requires pahole >= 1.16.

Brendan Gregg's Blog: ***BPF binaries: BTF, CO-RE, and the future of BPF perf tools*** ***https://www.brendangregg.com/blog/2020-11-04/bpf-co-re-btf-libbpf.html***

- BPF_MAP_TYPE_RINGBUF map, kernel should be **v5.8 and later**.

Dependencies for eBPF-based tool compilation/deployment

- Bpftrace and libbpf based tool requiring customers to install the **LLVM, Clang**, and kernel header dependencies for compilation.

- BCC tools use Python, and needs LLVM as well at runtime, it only targets 64-bit architecture.

- Ply is designed for embedded systems, only depends on LIBC, but it needs even more kernel configuration options. This requires the customer to enable more options in the kernel for the product on the field. In case of higher workload, there are event lost.

- **BTF** (BPF Type Format) and **CO-RE** (BPF Compile-Once Run-Everywhere) eliminate above dependencies at runtime but require the customer to use the latest kernel version and enable more options in the configuration.

Challenges of deploying eBPF-based tracing in embedded systems

- It is challenging to build eBPF-based tracing applications that are compatible with various Linux distributions in embedded systems. Especially for embedded systems in the field, the kernel is relatively not up-to-date and the kernel configuration does not fully enable eBPF.

- For embedded systems, we have no control over the environment of the target system. It is not possible to provide a unified/universal eBPF tracing application to cater to various Linux distributions with different kernel versions, kernel configurations.

- To resolve all dependencies, it is best to compile your eBPF-based application on the target platform to ensure that the tool can run with 100% guarantee. This is not possible for embedded systems.

Libtracefs, libtraceevent

Libtracefs is extracted out from trace-cmd, which allows programs to have an API to access the tracefs directory.

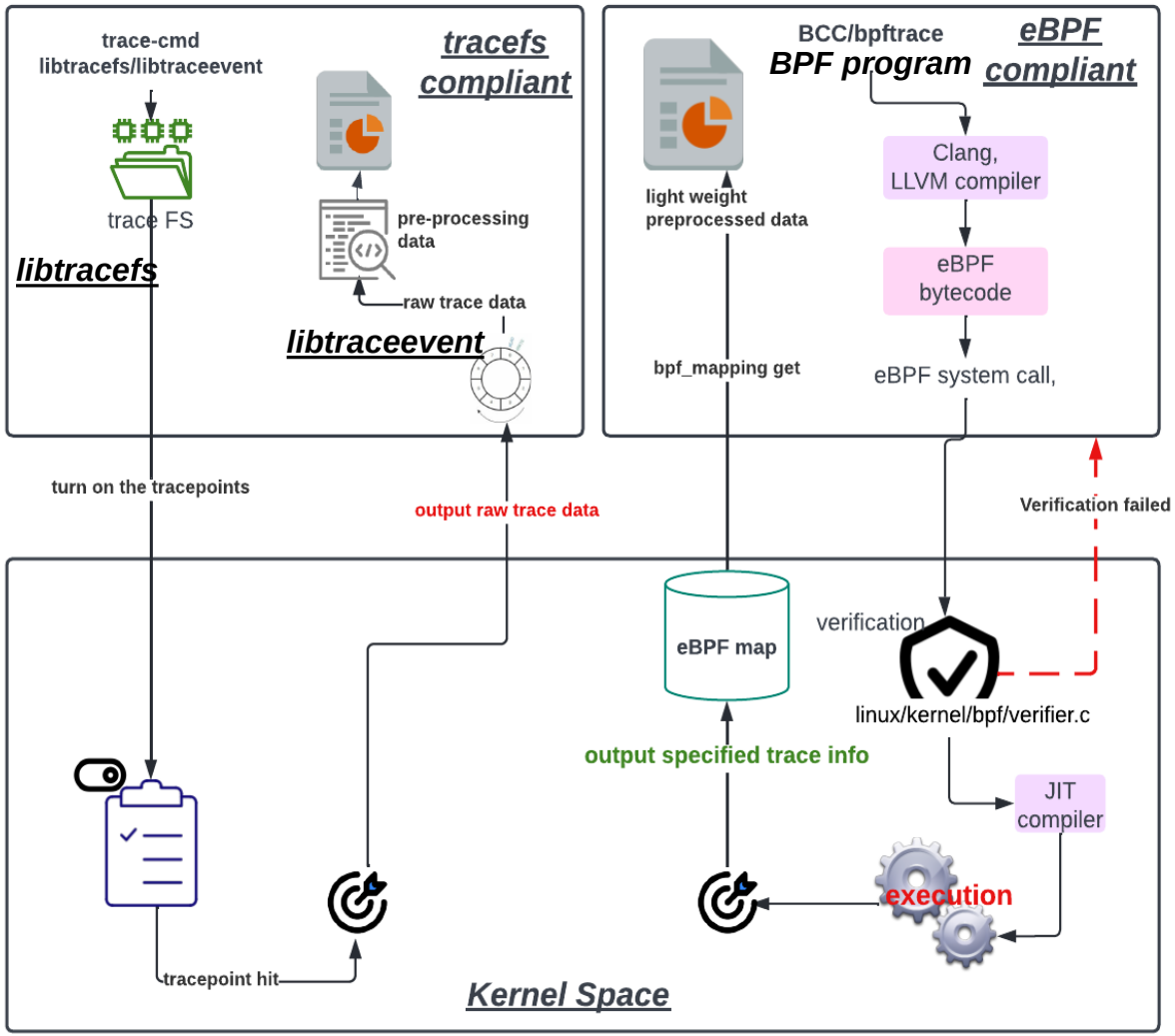
Libtraceevent library provides APIs to access kernel tracepoint events, located in the tracefs file system under the events directory, which provides APIs to be used to parse raw trace event formats.

Kernel configuration for Libtracefs, libtraceevent

- The kernel configuration required is only CONFIG_FTRACE=y:

```
CONFIG_FTRACE:

Enable the kernel tracing infrastructure.

Symbol: FTRACE [=y]
Type  : bool
Defined at kernel/trace/Kconfig:163
  Prompt: Tracers
  Depends on: TRACING_SUPPORT [=y]
  Location:
    Main menu
      -> Kernel hacking
```

Libtrace/libtraceevent (tracefs) and eBPF based trace systems data flow



- Libtracefs/libtraceevent based trace project enables tracepoint events through libtracefs, and then use APIs provides by Libtraceevent to get the trace data. But the data is outputed by kernel through a ringbuffer. Also, the libtraceevent will get the raw trace data. But user can use libtraceevent to filter data.

- eBPF is not used directly but indirectly via eBPF program, that co-exists with user-space C code in the eBPF compliant front-end tool. It should be compliled by LLVM to eBPF bytecode, and then can be loaded to the Linux kernel through eBPF system call. The trace data is pre-processed by eBPF program and will be print to the eBPF maps shared with userspace program.

Trace with libtracefs & libtraceevent

Configure tracefs with libtracefs API

- 100+ API make tracefs operation more convenient

trace event parse preparation

- Allocate and initialize tep (trace event parse namespace)
- Allocate kbuffer
- Load trace event format

Enable trace events, and register event handler

- Choose the Tracepoints that you want to get data sample from
- Event handler will be called while reading trace_pipe_raw, extract specific data from the kbuffer

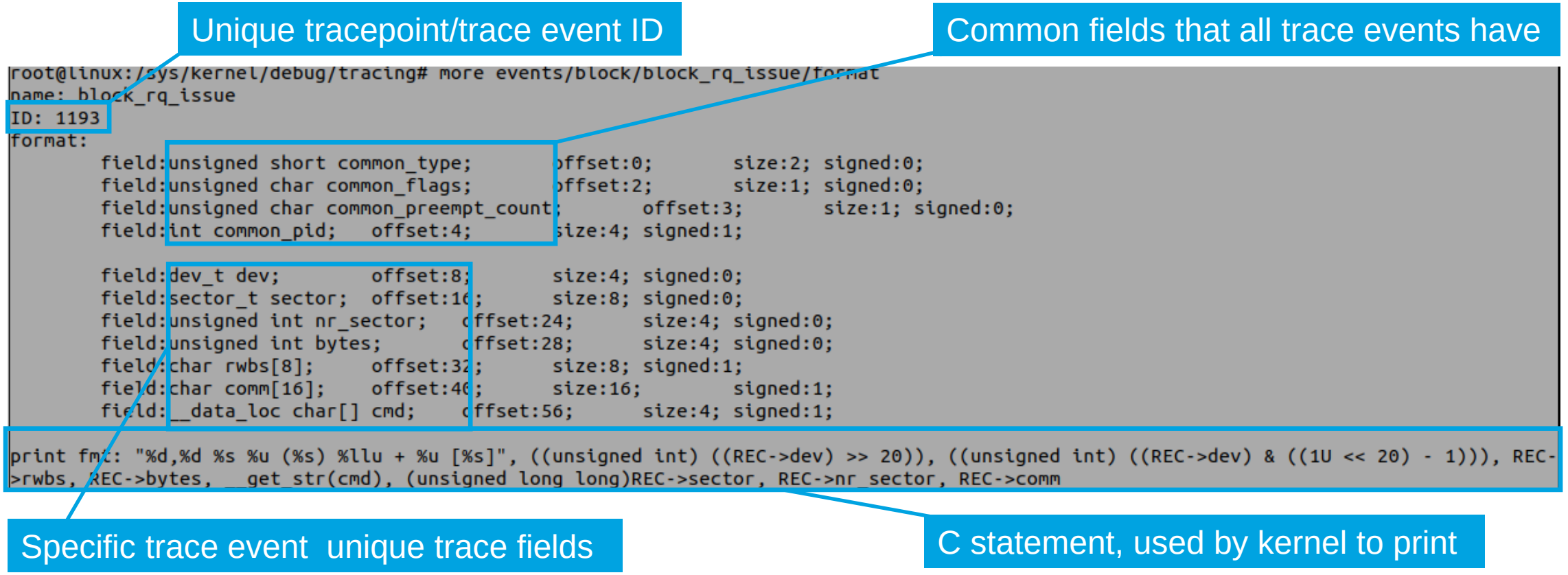
Read trace_pipe_raw

- The event handler will filer and parse the trace raw data

Configure tracefs with libtracefs API

```
#include <tracefs.h>

char *tracefs = NULL;
tracefs = tracefs_tracing_dir();

ret = tracefs_event_enable(NULL, "block", "block_rq_issue");
if (ret < 0 && !errno) {
...
}

if (tracefs_trace_on(NULL)) {
...
}

//read and extract your trace data with libtraceevent

tracefs_trace_off(NULL);
tracefs_event_disable(NULL, NULL, NULL);
```

Trace event formal file



- It is a sketch of binary raw data in the kernel ring buffer and explains how to read each trace item from each trace event raw data.
- libtraceevent mainly parses the format file of each event, which is convenient for parsing the original data, because the format file describes the binary layout of the original event data in kbuf.

Parse trace raw data with libtraceevent API


```
#include <traceevent/event-parse.h>
#include <traceevent/kbuffer.h>
#include <traceevent/trace-seq.h>
#include <tracefs.h>

struct tep_handle *tep;
struct kbuffer *kbuf;
char *buf; int size;

tep = tep_alloc();
…..
buf = read_file("/sys/kernel/tracing/events/header_page", &size);
Ret = tep_parse_header_page(tep, buf, size, sizeof(unsigned long));
…

ret = tracefs_load_event_format(tep, tracefs, "block", "block_rq_issue");
…

ret = tep_register_event_handler(tep, -1, "block", "block_rq_issue", block_rq_issue_handler, NULL);
…
cpus = sysconf(_SC_NPROCESSORS_ONLN);

while(1) {
…
    for (i = 0; i < cpus; i++) {
        char *raw_data;
        ret = asprintf(&raw_data, %s/per_cpu/cpu%d/trace_pipe_raw", tracefs, i);

        read_raw_buffer(tep, i, raw_buf);
        …..
        free(raw_data);
    }
}
```

Allocate tep(trace event parser namespace)

Load and parse header page format

load specific trace event data formant

register event handler

Get cpus

Read raw trace data

Read_raw_buffer()

```
static void read_raw_buffer(struct tep_handle *tep; int cpu, const char *raw_path)
{
     struct trace_seq s;
    char buf[page_size];
    int fd;
    int r;
    struct tep_record record;

    fd = open(raw_path, O_RDONLY | O_NONBLOCK);                  //open trace_pipe_raw
     ….
    while ((r = read(fd, buf, page_size)) > 0) {                 // read one page
        kbuffer_load_subbuffer(kbuf, buf);                       // load one page  to kbuffer
        record.cpu = cpu;
        for (;;) {
            record.data = kbuffer_read_event(kbuf, &record.ts);          // get one event trace data
            if (!record.data)
                  break;
          trace_seq_init(&s);
          tep_print_event(tep, &s, &record, “%s", TEP_PRINT_TIME);     //print record, and trigger handler
         kbuffer_next_event(kbuf, &ts);                                  // move to next one

         }
     }
 ….
   close(fd);
}
```

Event handler

```
static int block_rq_issue_handler(struct trace_seq *s, struct tep_record *record,
struct tep_event *event, void *context)    {
char *comm;
unsigned int dev;
unsigned int sector;
unsigned int nr_sector;
char *rwbs;
unsigned long long  ts;
unsigned long long pid  = 0;
int len;
Int cpu;

ts = record->ts;
cpu = record->cpu;
/*  extract comm */
if (!tep_get_common_field_val(NULL, event, "common_pid", record, &pid, 1)) {
        comm = tep_get_field_raw(NULL, event, "comm", record, &len, 0);
    …
}
/* dev */
if (!tep_get_field_val(NULL, event, "dev", record, &dev, 0)) {
}
/* LBA */
if (!tep_get_field_val(NULL, event, "sector", record, &sector, 0)){
}
/* length */
if (!tep_get_field_val(NULL, event, "nr_sector", record, &nr_sector, 0)){
}
/* operation */
rwbs = tep_get_field_raw(NULL, event, "rwbs", record, &len, 0);

printf("CPU %d, TS %llu.%llu, pid %lld, comm %s, %d:%d, LBA %x, sectors %d, %s\n",
cpu, ts/1000000000, ts%1000000000,  pid, comm, MAJOR(dev), MINOR(dev), sector,
nr_sector, rwbs);
…
…
}
```

- Note: Event handler is not mandatory, you can extract data sample in ***read_raw_buffer(…)***

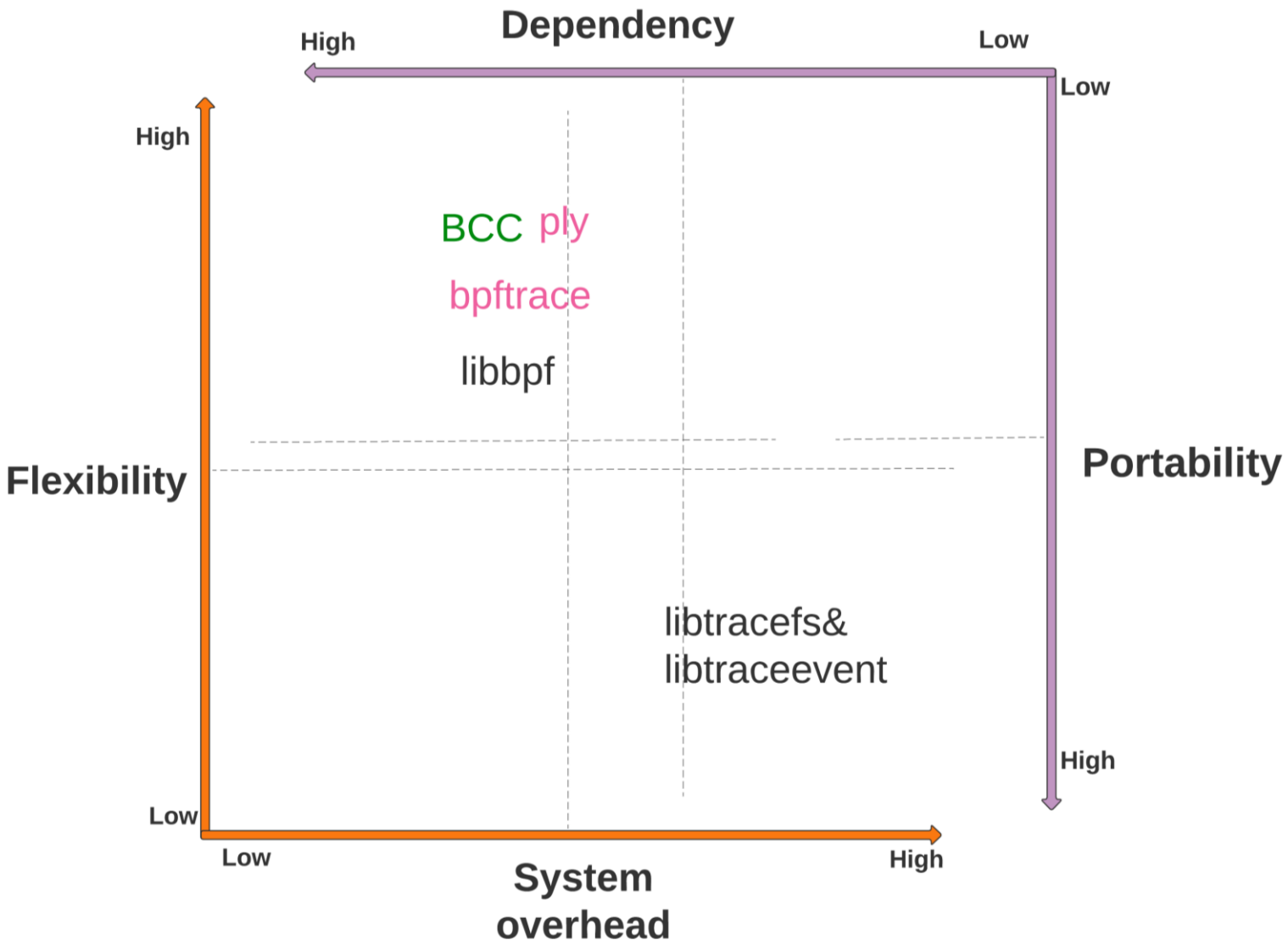
output:

```
CPU 2, TS 143753.925698664, pid 557502, comm kworker/u8:11, 8:16, LBA 1a5652a0, sectors 8, WM
CPU 2, TS 143753.925701933, pid 557502, comm kworker/u8:11, 8:16, LBA 1a565790, sectors 8, WM
CPU 2, TS 143753.925702751, pid 557502, comm kworker/u8:11, 8:16, LBA 1a565c38, sectors 8, WM
CPU 2, TS 143753.925703380, pid 557502, comm kworker/u8:11, 8:16, LBA 1a569488, sectors 8, WM
CPU 2, TS 143753.925704041, pid 557502, comm kworker/u8:11, 8:16, LBA 1a5695c8, sectors 8, WM
CPU 1, TS 143754.180707910, pid 1276, comm jbd2/sda1-8, 8:0, LBA 1de08cd8, sectors 16, WS
CPU 1, TS 143754.182665489, pid 222, comm kworker/1:2H, 8:0, LBA 0, sectors 0, FF
CPU 0, TS 143754.183122440, pid 104, comm kworker/0:1H, 8:0, LBA 1de08ce8, sectors 8, WS
CPU 0, TS 143754.183224578, pid 104, comm kworker/0:1H, 8:0, LBA 0, sectors 0, FF
CPU 0, TS 143755.460688029, pid 557500, comm kworker/u8:9, 8:16, LBA bb8f930, sectors 8, W
CPU 2, TS 143756.228638623, pid 252, comm jbd2/sdb6-8, 8:16, LBA 1333d930, sectors 32, WS
CPU 2, TS 143756.229460588, pid 221, comm kworker/2:2H, 8:16, LBA 0, sectors 0, FF
CPU 0, TS 143756.242354666, pid 104, comm kworker/0:1H, 8:16, LBA 1333d950, sectors 8, WS
CPU 0, TS 143756.242452515, pid 104, comm kworker/0:1H, 8:16, LBA 0, sectors 0, FF
CPU 1, TS 143756.740646950, pid 222, comm kworker/1:2H, 8:16, LBA 36db2c8, sectors 8, W
CPU 1, TS 143756.740656345, pid 222, comm kworker/1:2H, 8:16, LBA 42d70b8, sectors 32, W
CPU 1, TS 143756.740657620, pid 222, comm kworker/1:2H, 8:16, LBA 42d70e0, sectors 8, W
CPU 1, TS 143756.740659529, pid 222, comm kworker/1:2H, 8:16, LBA 42d70f0, sectors 40, W
CPU 1, TS 143756.740660474, pid 222, comm kworker/1:2H, 8:16, LBA 42d7120, sectors 40, W
CPU 1, TS 143756.740661575, pid 222, comm kworker/1:2H, 8:16, LBA 42d7150, sectors 8, W
CPU 1, TS 143756.740662373, pid 222, comm kworker/1:2H, 8:16, LBA 42d7160, sectors 24, W
CPU 1, TS 143756.740663154, pid 222, comm kworker/1:2H, 8:16, LBA 42d7180, sectors 16, W
CPU 1, TS 143756.740664023, pid 222, comm kworker/1:2H, 8:16, LBA 42d7198, sectors 24, W
CPU 1, TS 143756.740664854, pid 222, comm kworker/1:2H, 8:16, LBA 42d71d0, sectors 8, W
CPU 1, TS 143756.740665736, pid 222, comm kworker/1:2H, 8:16, LBA 42d71e8, sectors 16, W
CPU 1, TS 143756.740666572, pid 222, comm kworker/1:2H, 8:16, LBA 42d7238, sectors 8, W
```

Example source code:
https://github.com/beanhuo/libtracefs_libtraceevent_tools.git ***-b examples***
Note: The examples will be constantly updated and changed, the source code in github may differ from what is shown in the presentation, special thanks to Tzvetomir Stoyanov (VMware) for cleaning up the examples.

Micron

Comparison


Dependency
High
Low
Low
High
BCC
ply
bpftrace
libbpf
Flexibility
Portability
libtracefs&
libtraceevent
High
Low
Low
High
System
overhead

Conclusion

- eBPF is a game changer for tracing applications, based on which we can develop flexible and powerful tracing applications and obtain data samples from kernel space and user space. But for embedded systems, portability is key. Deploying eBPF-based tracing applications in embedded systems is difficult, especially when we want to trace customer systems to troubleshoot problems. BTF and CO-RE significantly improve the portability of eBPF-based tools in embedded systems but require customers to enable more kernel options.
  - It is hoped that customers can adopt the latest kernel version and enable the options required for eBPF in future designs
- Libtracefs / libtraceevent provide very rich APIs that allow us to get data samples from the kernel in a comfortable and confident way. But we need to read binary raw_data explicitly from each CPU ringbuffer, which is more expensive.
  - Is it possible to attach event handlers to the tracepoint itself and let the tracepoint filter the trace data, e.g., smart tracepoints?


Micron®

Reference

- https://lttng.org/docs/v2.13/#doc-what-is-tracing **The LTTng Documentation**
- https://blogs.oracle.com/linux/post/intro-to-bcc-1 **Intro to Kernel and Userspace Tracing Using BCC**
- https://nakryiko.com/posts/bpf-portability-and-co-re/ **BPF CO-RE (Compile Once – Run Everywhere)**

- Thanks to Tzvetomir Stoyanov (VMware) for the support and cleanup of the examples in Github